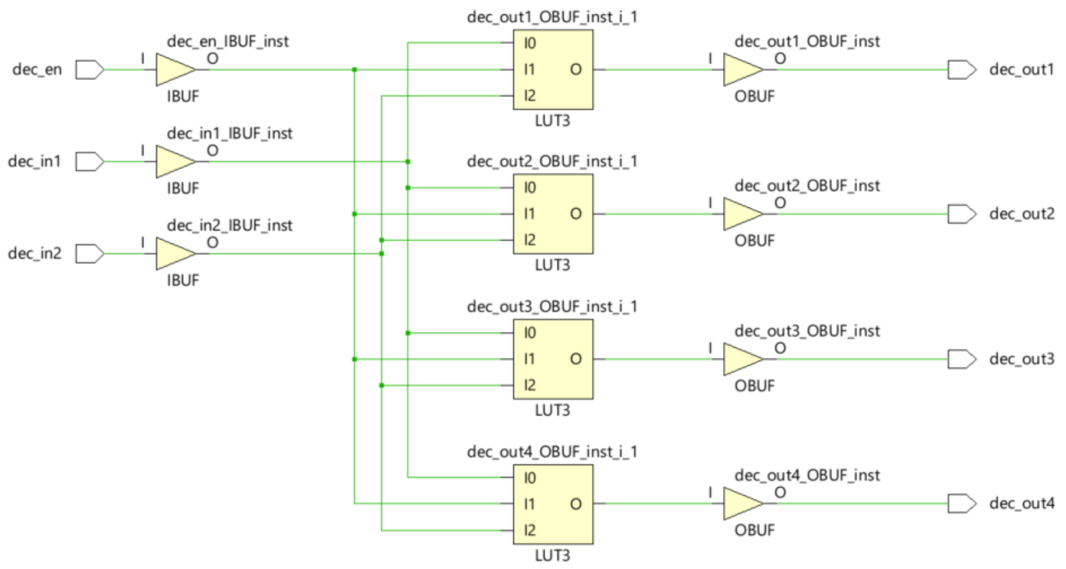
ECE3300L Lab1 Group H Report (Sherwin Sathish & Mohamed Hamida)

SCHEMATIC(2x4):



This schematic illustrates a 2x4 decoder with the enable as well. There are buffers for the inputs and outputs. There are 12 input and output ports that were addressed in the constraint file (with the inputs being switches and the outputs being LEDs) as shown below:

Xdc for 3x8:

```
##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { i[0] }]; #IO_L24N_T3_RS0_15
Sch=sw[0]
#set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }];
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { i[1] }];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]
#set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }];
#IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { i[2] }]; #IO_L12N_T1_MRCC_14
Sch=sw[4]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }]; #IO_L7N_T1_D10_14
Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { e }]; #IO_L17N_T2_A13_D29_14
Sch=sw[6]
## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { p[0] }]; #IO_L18P_T2_A24_15
Sch=led[0]
#set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { p[1] }]; #IO_L17N_T2_A25_15
Sch=led[2]
#set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14
Sch=led[3]
```

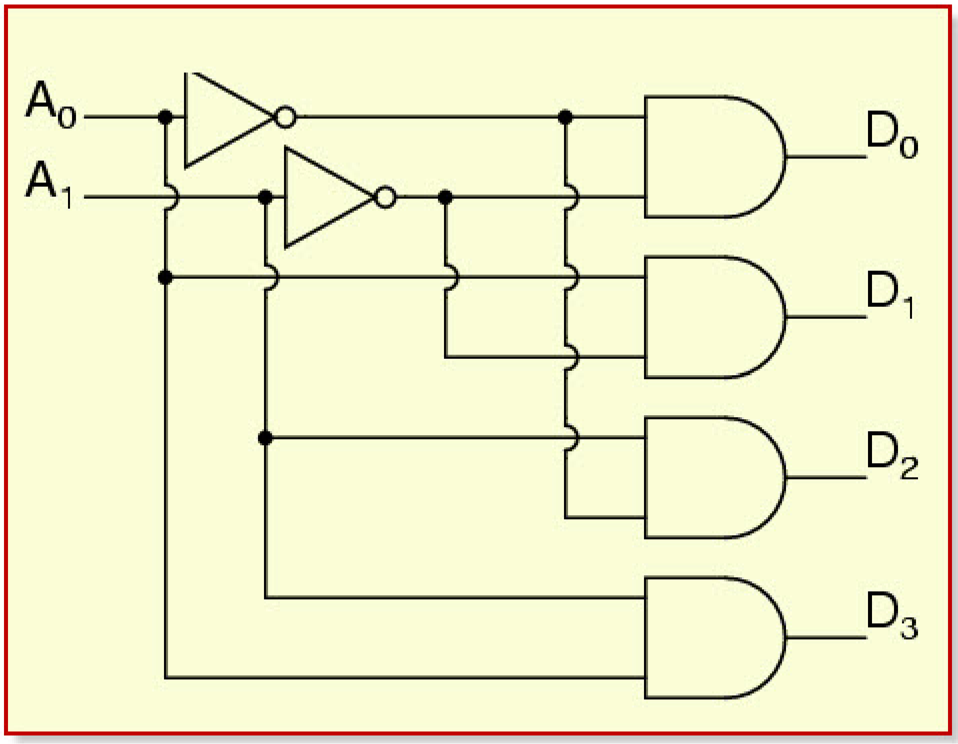
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { p[2] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { p[3] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { p[4] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
#set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { p[5] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { p[6] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { p[7] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
#set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

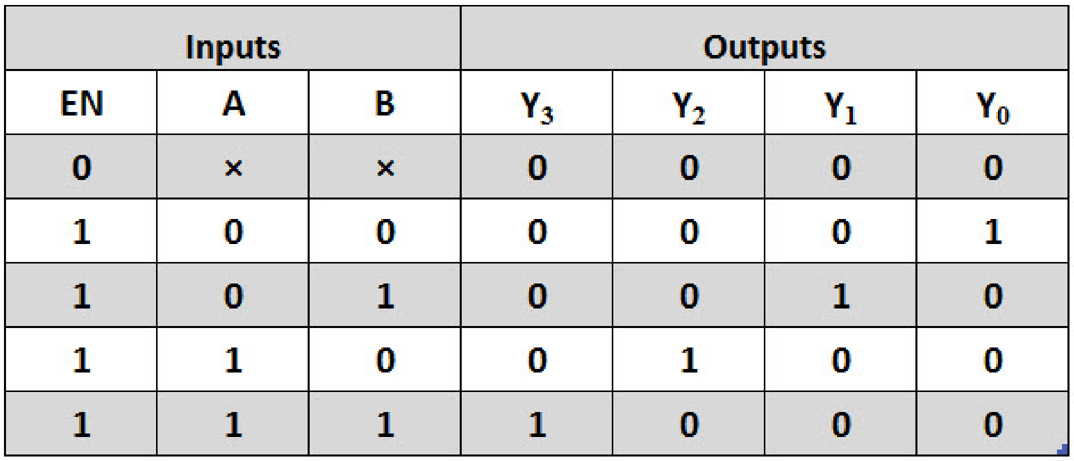
Code Detail:

DECO.v:

```
`timescale 1ns / 1ps
module DECO(
    input [1:0]in,
    input en,                    // Enable Input

    output [3:0]out
    );

 wire in0_neg, in1_neg;

 not(in0_neg, in[0]);
 not(in1_neg, in[1]);


 and(out[0],in1_neg,in0_neg,en);
 and(out[1],in1_neg,in[0],en);
 and(out[2],in[1],in0_neg,en);
 and(out[3],in[1],in[0],en);


Endmodule
```

The purpose of the DECO.v file is to create a 2x4 decoder by using not & and gates. In the module the input, enable, and output are defined. A wire is created for the initial inputs reaching the and gates. The schematic is created by the not() & and() functions. The _neg represents when the value for that particular input is 0. Ex: in0_neg=0 and i[0]=1. The not & and combination is based of the picture below:





Inputs			Outputs			
EN	A	B	Y_3	Y_2	Y_1	Y_0
0	×	×	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

There is an enable as well that is accounted for, the truth table is what determines what is inside the and functions.

DECO_tb.v:

```
module DECO_tb(

    );
    reg [1:0]in_tb;
    reg en_tb;

    wire [3:0]out_tb;

   DECO COMP_1   (
    .in(in_tb),
    .en(en_tb),                          // Enable Input
    .out(out_tb)                       // Output 1
    );

    initial
            begin:  TST1
               in_tb[0] = 1'b0;
               in_tb[1] = 1'd0;
               en_tb = 0;
             #10
               in_tb[0] = 1'b0;
               in_tb[1] = 1'd0;
               en_tb = 1;
               #10
                in_tb[0] = 1'd1;
                in_tb[1] = 1'd0;
                en_tb = 1;
             #10
                in_tb[0] = 1'd0;
                in_tb[1] = 1'd1;
                en_tb = 1;
            #10
                in_tb[0] = 1'd1;
                in_tb[1] = 1'd1;
                en_tb = 1;
            #1000
            $finish;
            end

endmodule
```

This code basically instantiates the new created variables by calling the previous class and creating a list of signals to connect to the component. Then with various different values the previous code is tested with these values.Hence, this is how the 2x4 is tested.

Xdc for 3x8:

```
##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { i[0] }]; #IO_L24N_T3_RS0_15
Sch=sw[0]
#set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW[1] }];
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports { i[1] }];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]
#set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }];
#IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { i[2] }]; #IO_L12N_T1_MRCC_14
Sch=sw[4]
#set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { SW[5] }]; #IO_L7N_T1_D10_14
Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { e }]; #IO_L17N_T2_A13_D29_14
Sch=sw[6]
## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { p[0] }]; #IO_L18P_T2_A24_15
Sch=led[0]
#set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { p[1] }]; #IO_L17N_T2_A25_15
Sch=led[2]
#set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]; #IO_L8P_T1_D11_14
Sch=led[3]
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { p[2] }]; #IO_L7P_T1_D09_14
Sch=led[4]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports { LED[5] }];
#IO_L18N_T2_A11_D27_14 Sch=led[5]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports { p[3] }];
#IO_L17P_T2_A14_D30_14 Sch=led[6]
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports { LED[7] }];
#IO_L18P_T2_A12_D28_14 Sch=led[7]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { p[4] }];
#IO_L16N_T2_A15_D31_14 Sch=led[8]
#set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports { LED[9] }];
#IO_L14N_T2_SRCC_14 Sch=led[9]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports { p[5] }];
#IO_L22P_T3_A05_D21_14 Sch=led[10]
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { LED[11] }];
#IO_L15N_T2_DQS_DOUT_CSO_B_14 Sch=led[11]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports { p[6] }]; #IO_L16P_T2_CSI_B_14
Sch=led[12]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports { LED[13] }];
#IO_L22N_T3_A04_D20_14 Sch=led[13]
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports { p[7] }];
#IO_L20N_T3_A07_D23_14 Sch=led[14]
#set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { LED[15] }];
#IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]
```

For the xdc, just uncomment the parts you're using and match the name with the .v file you want to test.

GenvarDECO.v:

```
`timescale 1ns / 1ps


module GenvarDECO #(parameter n=3)
(
   input[2:0]i,
   input e,
   output[7:0]p
);
  wire[1:0] temp;
genvar j;
generate
for(j=1; j<n; j=j+1)
   begin
       DECO UNIT (
         .in(i[1:0]),
         .out(p[(3+((j-1)*4)):((j-1)*4)]),
         .en(temp[j-1])
               );
   end
 endgenerate
  assign temp[0] = ~ i[2];
  assign temp[1]=i[2];
endmodule
```

This code creates a 3x8 decoder by calling the previous decoder twice and by using the temp wire to establish connections. To start off we create an input and output that matches the desired length of the decoder so 3x8 in this case. The temp wire is created because we are using a 2x4 decoder to connect to 2 2x4 decoders to have the end result be a 3x8. The for loop creates two decoders that use 2 new inputs and 8 new outputs, but there needs to be one more input ,which is the enable. Here the enable is the wire from the i[2] output of our original decoder to the two new decoders. This is why for the assign there's ~i[2] and i[2] because there are two different outputs for the original decoder as a whole for that i[2] input. This establishes the connection and creates the 3x8 decoder, the parameter is used to control the for loop.

GenvarDECO_tb.v:

```
module GenvarDECO_tb
  #(parameter s=8)
   (

   );

   reg [2:0] i_tb;
   reg e_tb;

   wire[7:0] p_tb;


  GenvarDECO
  #(
       .n(s)
  )
   GenvarDECO_TB
(
   .i(i_tb),
   .e(e_tb),
   .p(p_tb)

);
   initial
       begin:TSTB
       e_tb=1;
       i_tb=6;

       #100
       $finish;
       end


endmodule
```

This testbench tests the values for the previous nbit class. The parameter and the testbench variable are instantiated then tested.

4x16 decoder

```
`timescale 1ns / 1ps


module GenvarDECO #(parameter n=4)
(
    input[3:0]i,
    input e,
    output[15:0]p

);
  wire[3:0] temp;
  DECO UN (
            .in(i[3:2]),
            .out(temp[3:0]),
            //.en(1)
            .en(1)
          );

genvar j;
generate
for(j=0; j<n; j=j+1)
    begin


          DECO UNIT (
            .in(i[1:0]),
            .out(p[(3+((j)*4)):((j)*4)]),

            .en(temp[j])
          );

    end
 endgenerate
/
endmodule
```

```
`timescale 1ns / 1ps

module GenvarDECO_tb
   #(parameter s=16)
```

```
    (

    );

    reg [3:0] i_tb;
    reg e_tb;

    wire[15:0] p_tb;


  GenvarDECO
  #(
        .n(s)
   )
   GenvarDECO_TB
(
    .i(i_tb),
    .e(e_tb),
    .p(p_tb)

);
    initial
        begin:TSTB
        e_tb=1;
        i_tb[0]=0;
        i_tb[1]=0;
        i_tb[2]=0;
        i_tb[3]=0;
        #10
        i_tb[0]=1;
        i_tb[1]=0;
        i_tb[2]=0;
        i_tb[3]=0;
        #10
        i_tb[0]=0;
        i_tb[1]=1;
        i_tb[2]=0;
        i_tb[3]=1;
        #10
        i_tb[0]=1;
        i_tb[1]=0;
        i_tb[2]=1;
        i_tb[3]=1;
        #10
        i_tb[0]=1;
        i_tb[1]=1;
        i_tb[2]=1;
        i_tb[3]=1;

        #100
        $finish;
        end
```

```
endmodule
```

The logic for the 4x16 is basically the same as the 3x8 except there are double the outputs, but only one more input. So all the outputs from the original decoder have to be used as an enable for the 4 new decoders created. The for loop establishes the wire connections necessary for a 4x16 decoder.

Corner Cases/Error :

One of the corner cases we had was that we already built the initial 3x8 nbits code:

```
module GenvarDECO #(parameter n=3)
(
   input[2:0]i,
   input e,
   output[7:0]p
);
  wire[1:0] temp;
genvar j;
generate
for(j=1; j<n; j=j+1)
   begin
        DECO UNIT (
          .in(i[1:0]),
          .out(p[(3+((j-1)*4)):((j-1)*4)]),
          .en(temp[j-1])
                );
   end
 endgenerate
  assign temp[0] = ~ i[2];
  assign temp[1]=i[2];
endmodule
```

The corner case problem was that we needed to fix the code to test the parameters of the 4x16 decoder. This code uses wire temp to connect to the input of the first 2x4 decoder (i[2]), which then acts as an enable for the 2 new 2x4 decoders. That’s why .en(temp[j-1]) is there to make this connection. This code is only built for 3x8, but not 4x16. The correct code is shown below:

```
 module GenvarDECO #(parameter n=4)
(
   input[3:0]i,
   input e,
   output[15:0]p

);
  wire[3:0] temp;
  DECO UN (
          .in(i[3:2]),
          .out(temp[3:0]),
          .en(1)
```

```
        );
genvar j;
generate
for(j=0; j<n; j=j+1)
    begin
          DECO UNIT (
            .in(i[1:0]),
            .out(p[(3+((j)*4)):((j)*4)]),
            .en(temp[j])
          );
    end
 endgenerate
endmodule
```

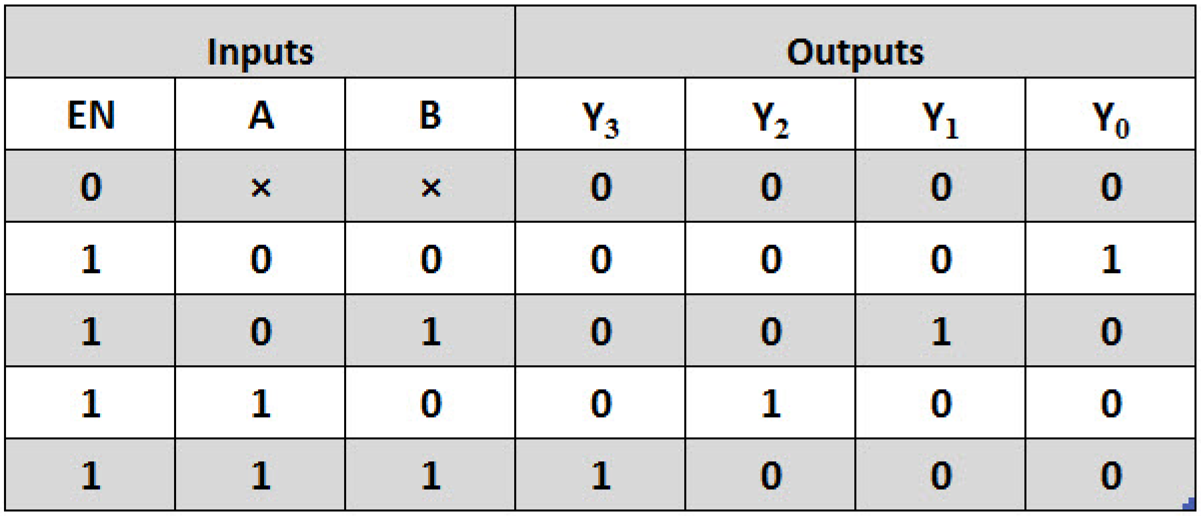
I used this code for the 4x16 and 3x8 because DECO UN calls the first 2x4 decoder, but it establishes the wire connection with the outputs so that it can be used as an enabler when the other decoders are being created in the for loop. The only issue I had with the old code is that the output repeats for every iteration of 8 outputs, so DECO UN eliminates the corner case of repeating.
Another corner case /error was the enable. The enable would manually have to be change from 0 to 1, instead of the testbench determining the value.

Ex:

```
module GenvarDECO #(parameter n=4)
(
    input[3:0]i,
    input e,
    output[15:0]p

);
  wire[3:0] temp;
  DECO UN (
            .in(i[3:2]),
            .out(temp[3:0]),
            .en(1)  // highlighted portion determines the enable
          );
genvar j;
generate
for(j=0; j<n; j=j+1)
    begin
          DECO UNIT (
            .in(i[1:0]),
            .out(p[(3+((j)*4)):((j)*4)]),
            .en(temp[j])
          );
    end
 endgenerate
endmodule
```

Decoder 2x4 Testbench:

The truth table of a 2x4 decoder is shown through waveforms with this particular testbench. The truth table that is being displayed is shown below.



Inputs			Outputs			
EN	A	B	Y_3	Y_2	Y_1	Y_0
0	×	×	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

A demo was made to show this in action:
https://www.youtube.com/watch?v=oU2y9_GVhOQ&t=50s

GenvarDECO Testbench "nbits":

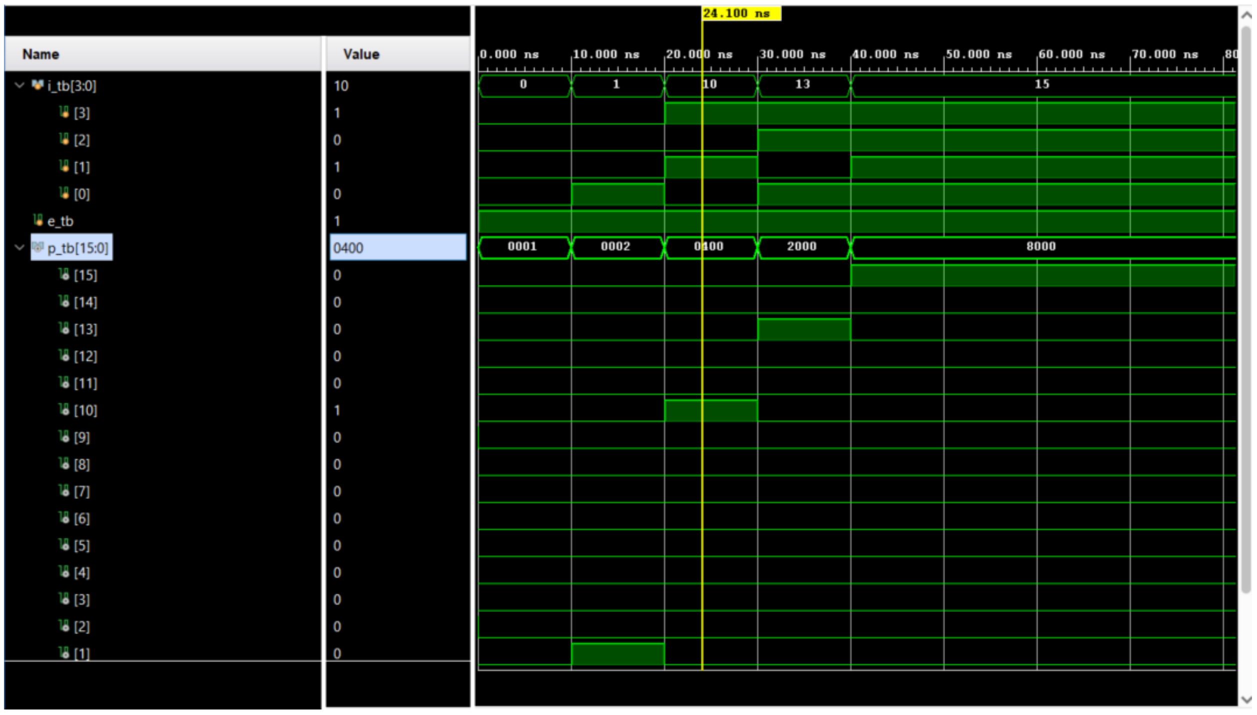
The truth table of a 3x8 decoder is shown through a for-generated loop with this particular testbench. The for-generated loop is shown below:

```
genvar i;
generate
for(j=1; j<n; j=j+1)
   begin
        DECO UNIT (
          .in(i[1:0]),
          .out(p[(3+((j-1)*4)):((j-1)*4)]),
          .en(temp[j-1])
                );

   end
 endgenerate
```

The purpose of the for loop is to create different decoders that connect back to the original decoder to create a 3x8 in this case.

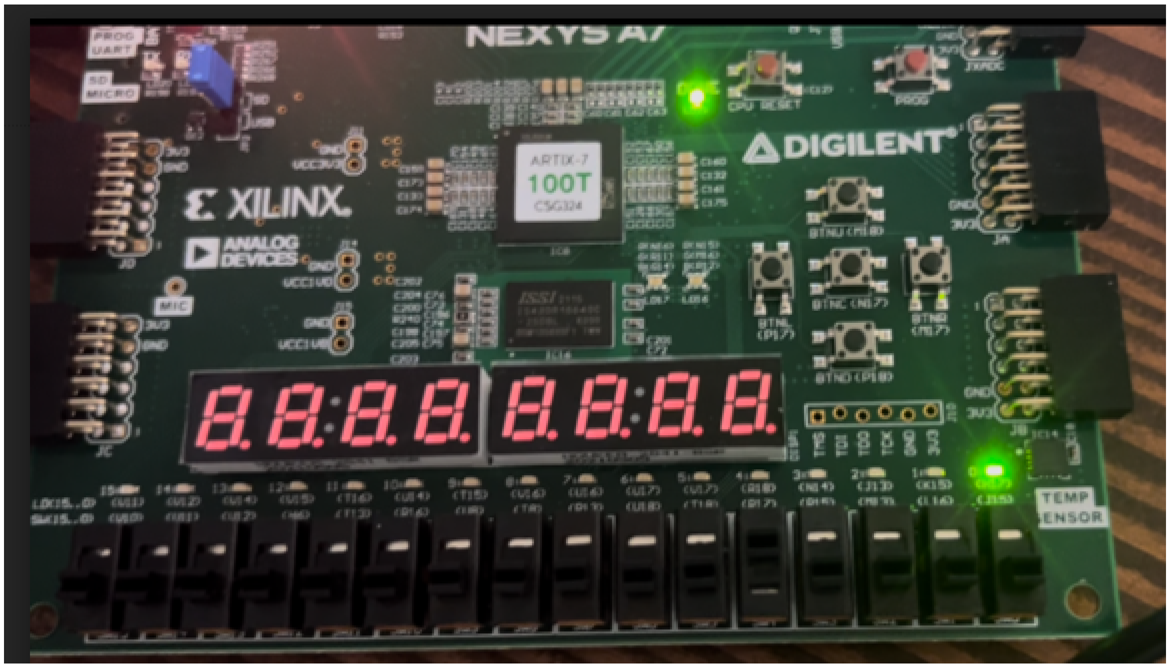
A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

<u>4x16 TB:</u>



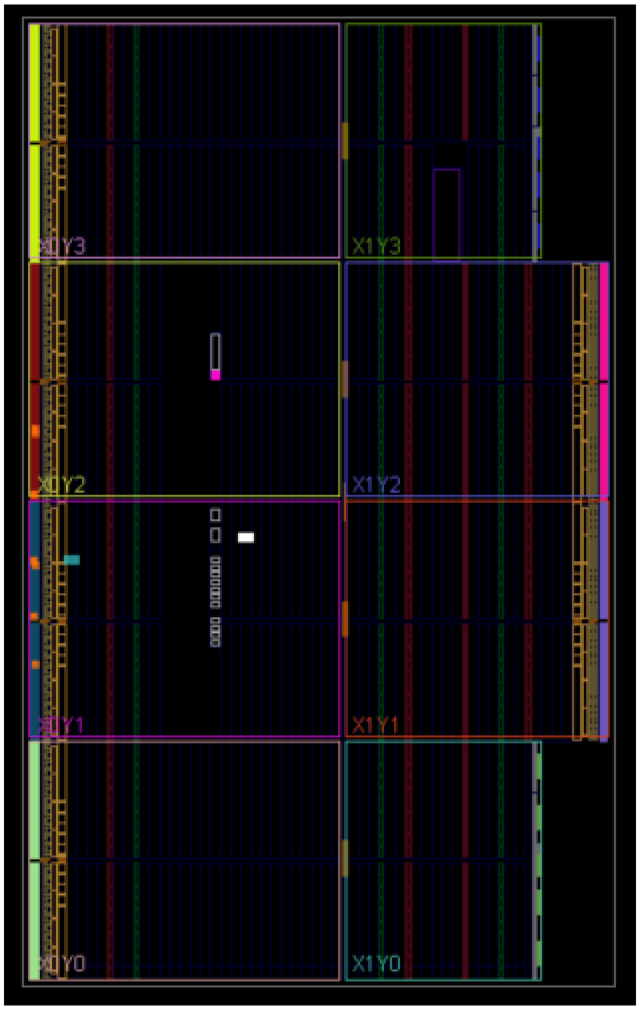
Name	Value
i_tb[3:0]	10
[3]	1
[2]	0
[1]	1
[0]	0
e_tb	1
p_tb[15:0]	0400
[15]	0
[14]	0
[13]	0
[12]	0
[11]	0
[10]	1
[9]	0
[8]	0
[7]	0
[6]	0
[5]	0
[4]	0
[3]	0
[2]	0
[1]	0

<u>NEXYS A7:</u>

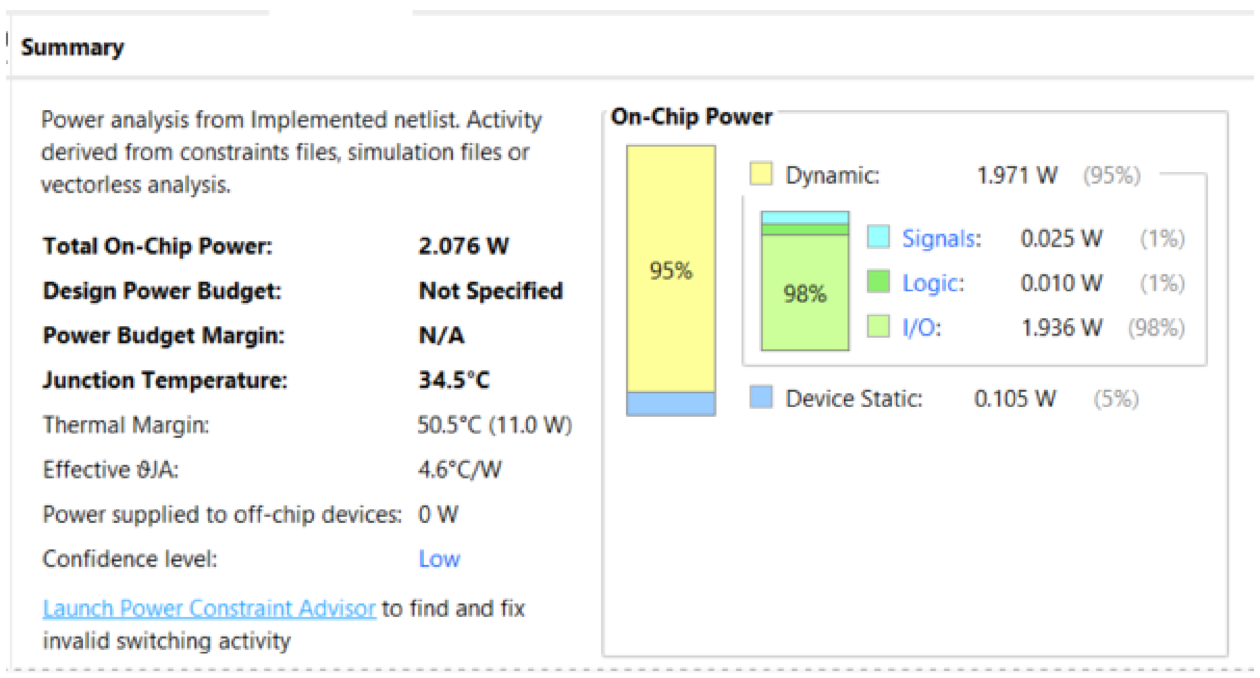


This is an example of the enable switch being on and the inputs being 0 for a 3x8 decoder and the LSB LED is on. This proves how the 3x8 decoder has been successfully programmed into the board.

IMPLEMENTED DESIGN:



POWER SUMMARY:



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

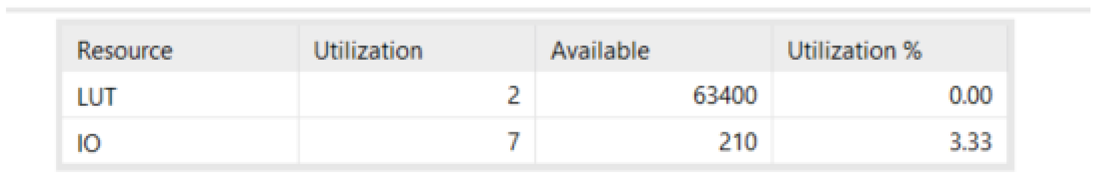
Total On-Chip Power:	**2.076 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**34.5°C**
Thermal Margin:	50.5°C (11.0 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

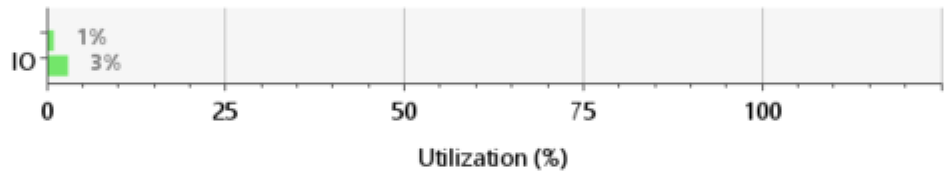
On-Chip Power

🟨	Dynamic:	1.971 W	(95%)
🟦	Signals:	0.025 W	(1%)
🟩	Logic:	0.010 W	(1%)
🟩	I/O:	1.936 W	(98%)
🟦	Device Static:	0.105 W	(5%)

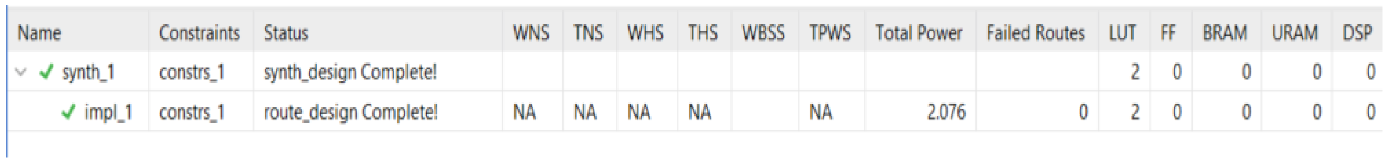
UTILIZATION:



Resource	Utilization	Available	Utilization %
LUT	2	63400	0.00
IO	7	210	3.33



RESOURCE USAGE:



Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!									2	0	0	0	0
✓ impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA		NA	2.076	0	2	0	0	0	0